AFRL-AFOSR-JP-TR-2016-0078

Binary CFG Rebuilt of self-modifying codes

Mizuhito Ogawa
JAPAN ADVANCED INSTITUTE OF SCIENCE AND TECHNOLOGY

10/03/2016
Final Report

Air Force Research Laboratory
AF Office Of Scientific Research (AFOSR)/ IOA
Arlington, Virginia 22203
Air Force Materiel Command

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
04-10-2016	Final	12 May 2014 to 11 May 2016

4. TITLE AND SUBTITLE
Binary CFG Rebuilt of self-modifying codes

5a. CONTRACT NUMBER

5b. GRANT NUMBER
FA2386-14-1-4050

5c. PROGRAM ELEMENT NUMBER
61102F

6. AUTHOR(S)
Mizuhito Ogawa

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
JAPAN ADVANCED INSTITUTE OF SCIENCE AND TECHNOLOGY
1-1 ASAHIDAI NOMI
ISHIKAWA, 21421016 JP

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
AOARD
UNIT 45002
APO AP 96338-5002

10. SPONSOR/MONITOR'S ACRONYM(S)
AFRL/AFOSR IOA

11. SPONSOR/MONITOR'S REPORT NUMBER(S)
AFRL-AFOSR-JP-TR-2016-0078

12. DISTRIBUTION/AVAILABILITY STATEMENT
A DISTRIBUTION UNLIMITED: PB Public Release

13. SUPPLEMENTARY NOTES

14. ABSTRACT
Modern malware extensively applies self-modifying obfuscation techniques, e.g., self-decryption and mutation, which are often automatically prepared by packers. Their aim is to confuse control structures and bypass commercial anti-virus software based on binary signatures. A popular method in industry to analyze malware is a dynamic analysis in a sand-box. Alternatively, we apply a hybrid method combining concolic testing (dynamic symbolic execution) and Windows API stubs by external executions. They are implemented as BE-PUM (Binary Emulation for Pushdown Model generation), which shows strong disassembly ability (control flow graph generation) at the cost of relatively heavy execution. For instance, BE-PUM automatically detects the destination server of EMDIVI, which caused huge information leak from Japanese governmental pension fund in 2015.
The first year of the project, we developed BE-PUM from a preliminary prototype, which supports 15 x86 instructions and no Windows APIs, to support 100 x86 instructions and 400 Windows APIs. These x86 binary emulation and Windows API stubs are manually prepared. We also perform experiments on several thousand real malware to evaluate BE-PUM design. The second year, we worked on several topics. (1) Multi-threading for faster processing, (2) Automatic Windows API stub generation from natural language specification provided by MSDN, (3) Loop invariant generation for binary programs, and (4) Packer identification that is used when malware is made, (1)-(3) enhance BE-PUM more complete and efficient, and (4) shows that BE-PUM can precisely detect and classify individual obfuscation techniques. Next step will be to analyze contamination techniques.
This project is performed under collaborations with Ho-Chi-Minh University of Technology (Vietnam) and LOIRA, University of Lorraine (France).

15. SUBJECT TERMS
Malware

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			SERNA, MARIO
Unclassified	Unclassified	Unclassified	SAR	7	19b. TELEPHONE NUMBER *(Include area code)* 315-227-7007

Standard Form 298 (Rev. 8/98)
Prescribed by ANSI Std. Z39.18

Final Report for AOARD Grant FA2386-14-1-4050

"**Binary CFG Rebuilt of self-modifying codes**"

August 11 2016

Name of Principal Investigators (PI and Co-PIs): Mizuhito Ogawa
- e-mail address : mizuhito@jaist.ac.jp
- Institution : Japan Advanced Institute of Science and Technology
- Mailing Address : 1-1 Asahidai Nomi Ishikawa, 923-1292 Japan
- Phone : +81-761-51-1247
- Fax : +81-761-51-1149

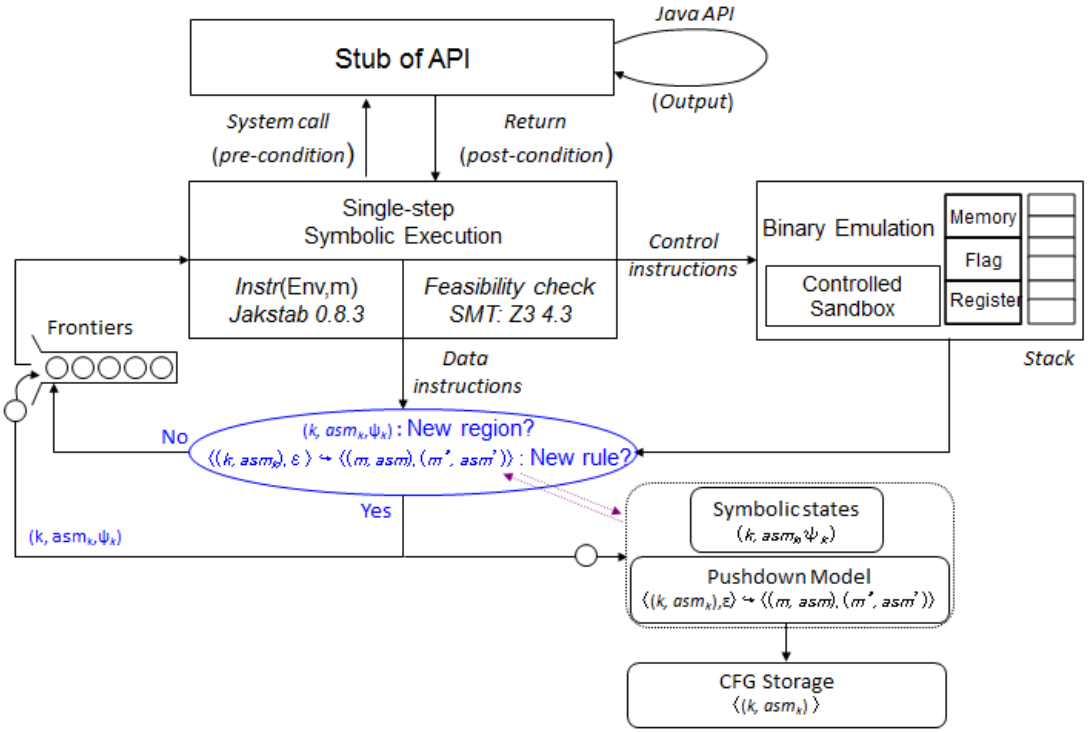
Period of Performance: April/1/2014 – May/11/2016

Abstract: Modern malware extensively applies self-modifying obfuscation techniques, e.g., self-decryption and mutation, which are often automatically prepared by packers. Their aim is to confuse control structures and bypass commercial anti-virus software based on binary signatures. A popular method in industry to analyze malware is a dynamic analysis in a sand-box. Alternatively, we apply a hybrid method combining concolic testing (dynamic symbolic execution) and Windows API stubs by external executions. They are implemented as BE-PUM (Binary Emulation for Pushdown Model generation), which shows strong disassembly ability (control flow graph generation) at the cost of relatively heavy execution. For instance, BE-PUM automatically detects the destination server of EMDIVI, which caused huge information leak from Japanese governmental pension fund in 2015.
The first year of the project, we developed BE-PUM from a preliminary prototype, which supports 15 x86 instructions and no Windows APIs, to support 100 x86 instructions and 400 Windows APIs. These x86 binary emulation and Windows API stubs are manually prepared. We also perform experiments on several thousand real malware to evaluate BE-PUM design. The second year, we worked on several topics. (1) Multi-threading for faster processing, (2) Automatic Windows API stub generation from natural language specification provided by MSDN, (3) Loop invariant generation for binary programs, and (4) Packer identification that is used when malware is made, (1)-(3) enhance BE-PUM more complete and efficient, and (4) shows that BE-PUM can precisely detect and classify individual obfuscation techniques. Next step will be to analyze contamination techniques.
This project is performed under collaborations with Ho-Chi-Minh University of Technology (Vietnam) and LOIRA, University of Lorraine (France).

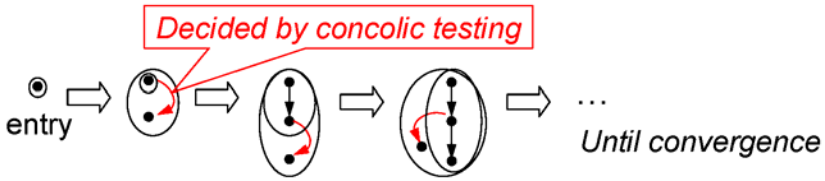
Introduction: Malware is an obvious threat. Our ultimate goal is the malware classification by techniques, including the family tree of the evolutionary relationship, rather than malware detection. Malware consists of three steps, obfuscation to bypass commercial anti-virus software, contamination of a system to spread, and malicious behavior like information leakage. Symantec Norton developers confessed on May 2014 that antivirus software can detect only 45% of malware due to recent obfuscation techniques, and it is said that more than 80% of recent malware is made by packers, which automatically inserts obfuscation codes.
This project focuses on the obfuscation techniques. There are two major analysis techniques, popular dynamic analyses in sand boxes, and static analyses mostly in academia, e.g., JakStab and McVeto. Our method is in between, a hybrid analysis combining concolic testing (dynamic symbolic execution) and external execution of Windows API as stubs, which is implemented as BE-PUM (Binary Emulator for PUshdown Model generation).
Our choice for binary emulation is the user process level, aiming a light-weighted implementation and the flexibility for detecting trigger-based behaviors. As a result, BE-PUM obtains strong disassembly ability, e.g., automatic detection of the destination server of EMDIVI, which caused huge information leakage from Japanese governmental pension funds in 2015.
This project is under collaboration with Ho-Chi-Minh City University of Technology, Vietnam and LOIRA, University of Lorraine, France.

Experiment: The design of our binary code analyzer BE-PUM is the combination of concolic testing (dynamic symbolic execution) in a user-process level binary emulator and Windows API stubs

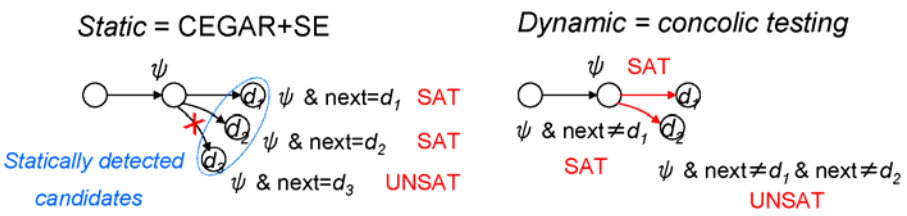
in real Windows environments. This is based on the independence of an external action of a stub, which is often observed in a heterogeneous system, e.g., Java web applications querying external SQL servers. In the symbolic execution, this observation enables the post-condition of a stub kept unchanged from the pre-condition. The figure below illustrates the architecture of BE-PUM, in which an x86 instruction is executed in the user-level binary emulator to decide the next destination of an indirect jump. A Windows API stub keeps the pre/post conditions in symbolic execution unchanged, and updates the environment with an external execution of Windows API, called in JNA environment.



The model (CFG) generation is in an on-the-fly manner. X86 binary is executed by stepwise interpretation, since an x86 instruction varies its length and where the next instruction starts is decided by this interpretation.

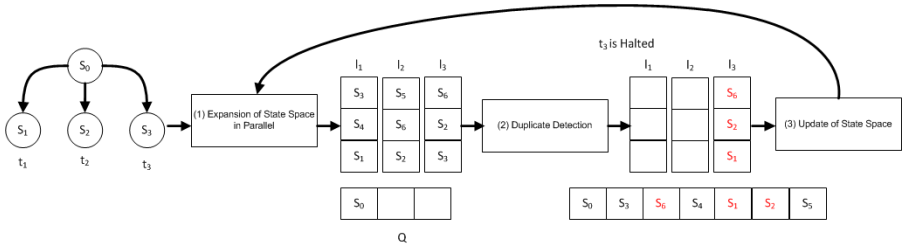


If the current x86 instruction is a data instruction, such as INC and MOV, the next instruction starts from the next bit. However, if it is a control instruction like conditional jumps, the next instruction is decided dynamically by computing the next address. To decide next destinations of an indirect jump, there are two methods: static and dynamic. The former, first enumerates possible next destinations and check the feasibility one-by-one by satisfiability checking of its path condition. The latter is concolic testing, i.e., simply generates a test input as a satisfiable instance of the path condition and performs binary emulation, until no more destinations are found (i.e., the path condition becomes unsatisfiable). The satisfiability checking is typically done by an SMT solver, such as Z3.

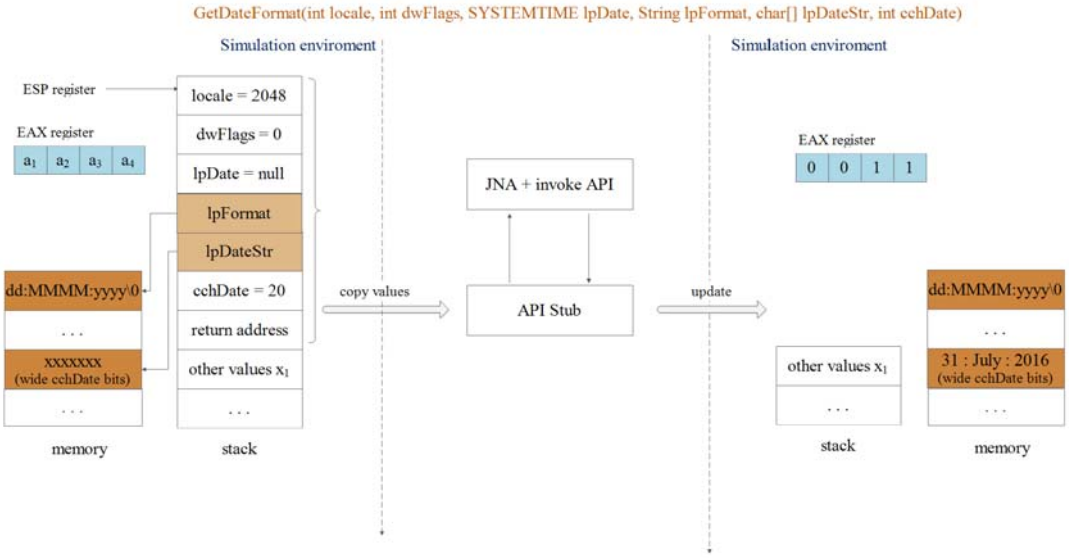


A preliminary prototype of BE-PUM was developed in 2013 (*M.H.Nguyen, T.B.Nguyen, T.T.Quan, M.Ogawa, A hybrid approach for control flow graph construction from binary code*, APSEC 2013), which supports 15 x86 instructions and no Windows APIs. The first year of the project, we developed BE-PUM such that current BE-PUM supports 100 x86 instructions (among 1000) and 400 Windows APIs (among 4000). These x86 binary emulation and Windows API stubs are manually prepared.
We also perform experiments on several thousand real malware taken from VX-Heaven virus database and some supplied from LORIA, University of Lorraine. BE-PUM shows strong disassembly ability beyond popular commercial disassemblers, e.g., IDA Pro and Capstone. For instance, the differences of disassembly results between BE-PUM and IDA Pro identify the location of obfuscation codes [c1,o1]. Another notable example is EMDIVI, of which the destination server of information leakage is automatically detected by the disassembly of BE-PUM.

The second year, we worked on several topics. (1) Multi-threading for faster processing, (2) Automatic Windows API stub generation from natural language specification provided by MSDN, (3) Loop invariant generation for binary programs, and (4) Packer identification that is used when malware is made. (1)-(3) enhance BE-PUM more complete, and (4) shows that BE-PUM can precisely detect and classify individual obfuscation techniques.
(1) BE-PUM provides strong disassembly ability beyond popular commercial disassemblers; however, its execution is quite heavy. We tried multi-threaded implementation, which shows almost linear growth of the efficiency when the number of CPUs are up to 4 [c2]. To reduce communication cost among CPUs, we introduce local lists of tasks of next control destination explorations, which are managed by hash tables.



(2) Currently, BE-PUM supports more than 100 x86 instructions and 400 Windows APIs. They exist more than 1000 x86 instructions and 4000 Windows APIs, which show the engineering difficulty of a manual implementation. We observe that both x86 and Windows API are executable (thus testable), and their specification in natural languages mostly follows to fixed formats, e.g., API specification at MSDN (MicroSoft Developer Network). Furthermore, Windows API stub requires only limited information, since JNA executes API in real Windows environments. We focused on automatic Windows API stub generation, and collected 1800 descriptions of APIs mostly from MSDN. With the aid of natural language processing, we successfully generated 1200 stubs. The figure below shows a generated API stub for “GetDataFormat”. This work is under preparation for publication.



(3) As an algorithm optimization of the symbolic execution, we investigated how to guarantee the termination of loop unfolding and how to reduce the cost of loop unfolding. We first tried to distinguish loops with fixed and unbounded numbers of iterations, by statically analyzing *constant propagation*. Then, loops with fixed numbers of iterations are safely unfolded until the bound [o2]. The ultimate solution for both termination and optimization of loop unfolding is the loop invariant generation to skip symbolic execution of further loop unfolding. Although an automatic loop invariant generation is recently well-investigated in high-level languages like C, binary code has additional difficulties. For instance, there are no clear loop/recursion statements, e.g., *do-while* and *call/return.*, Worse, binary code arbitrary inserts push/pop operations and memory updates. We restrict our attention to self-decryption loops, in which stack operations and arithmetic operations are observed quite independent. We apply Difference logic on stack operations and Presburger arithmetic on arithmetic operations for loop invariant generation. This work is on-going.

(4) BE-PUM obtains strong disassembly ability, but it is not easy to evaluate. We apply the disassembly result on packer identification, i.e., which packer is used to generate a given malware. We first formally define each obfuscation technique, which appear in various surveys with rather informal natural language descriptions. For instance, *Code Chunking* is formally specified as 3 jump instructions in 20 bytes. These numbers are carefully chosen by manual observation. We selected popular 10 packers, e.g., UPX, Yoda, TElock, and PEcompat, and observed that each packer inserts obfuscation techniques in a specific order. The table below shows how a small binary code is packed and what kinds of obfuscation techniques are inserted (in which the number indicates certain obfuscation technique, e.g., 0 means *anti-debugging*, 1 means *checksumming*, and 2 means *code chunking*. The sequences are quite consistent among other small packed binary codes.

UPX v3.0	4-7-8-4-4-4-4-1-4-4-4-4-7-4-5-4-7-4-7-4-7-4-7-4-7-4-4-4-4-4-4-3-3-12-4-12-3-4-4-3-4
ASPack v2.12	4-7-3-3-3-3-10-4-3-10-4-7-7-4-7-7-4-8-1-4-4-7-7-7-4-7-7-7-7-7-4-4-4-7-4-4-4-4-4-4-4-4-4-6-4-4-4-7-7-4-4-4-5-4-4-5-4-7-4-4-7-7-7-4-7-4-4-7-4-7-7-7-4-7-4-4-4-7-7-7-4-4-5-4-7-7-4-4-4-7-7-7-7-4-4-6-4-7-7-2-5-4-4-4-4-4-7-7-4-4-3-4-7-10-7-7-7-10-3-7-3-4-3-4-5-4-3-4-4-3-3-6
FSG v2.0	7-7-3-3-7-3-3-8-7-5-3-3-3-6-4-3-4-3-1-3-3-12-12-3
nPack v1.0	3-3-12-3-3-10-6-5-4-1-4-4-3-4-4-4-3-10-4-7-7-7-7-8-7-7-7-7-4-4-4-4-7-7-4-4-4-4-4-3-7-4-5-10-10-4-4-3-12-4-3-4-4-5-4-12-4-6
PECompact 2.0x	9-9-7-7-4-7-7-9-4-3-10-7-7-7-3-7-7-7-6-1-4-4-7-8-5-7-7-4-7-3-4-4-7-7-7-7-4-4-3-7-3-12-4-4-12-3-10-4-4-4-5-4-4-4-7-4-3-6-7-7-4-4-7-5-7-7-4-4-4-4-4-4-4-5-4-4-4-4-4-4-4-4-4-4-4-5-4-4-4-7-4-4-4-3-3-4-3-4-4-4-5-4-4-4-4-3-3-4-4-4-4-4-3-4-3-3-5-7-3
PEtite v2.1	9-9-7-3-7-3-4-7-3-4-7-4-4-7-3-4-7-4-7-4-7-4-7-4-7-4-7-8-7-7-7-4-4-7-4-4-4-4-4-4-1-4-4-4-4-4-4-4-7-4-4-4-4-4-7-4-4-4-4-4-4-4-4-4-4-4-4-4-4-5-4-4-4-4-4-4-5-4-5-4-4-4-4-4-4-4-4-4-4-4-4-4-4-4-4-5-4-7-7-4-4-4-4-4-4-4-4-4-4-4-4-5-4-4-4-5-4-4-4-6-4-4-4-4-7-5-7-4-5-4-4-7-7-7-7-7-7-7-7-7-9-3-4-7-7-3-5-7-4-4-4-7-4-4-4-4-5-7-5-9-7-7-4-4-4-4-3-3-4-4-4-4-7-7-4-7-4-7-4-7-4-4-4-4-7-7-4-7-7-4-5-7-4
uPack v0.36	4-4-4-8-4-4-4-3-4-4-4-4-4-4-8-4-4-4-4-4-8-4-4-13
yoda's Crypter 1.3	4-7-4-4-4-4-4-4-4-4-4-4-4-4-5-7-4-7-4-4-7-7-7-4-7-4-7-4-7-7-4-7-7-4-7-9-9-7-4-9-4-1-8-7-4-4-4-7-7-4-4-7-7-4-7-4-7-7-4-3-7-4-7-7-4-7-4-3-12-7-4-7-4-7-4-7-4-7-4-7-4-7-4-7-4-7-4-7-4-7-4-7-4-7-4-7-4-7-4-7-4-7-4-7-4-7-4-3-7-4-7-4-7-4-3-7-4-3-7-7-4-3-7-4-7-4-4-3-7-4-7-4-3-4-7-4-6-7-4-3-7-4-3-7-4-4-7-4-4-4-4-2-4-4-4-4-5-4-4-4-6-7-4-7-4-4-7-7-4-7-4-7-7-4-7-4-4-4-7-4-7-7-4-4-4-7-5-7-4-3-4-7-7-4-7-4-6-7-4-3-12-7-4-7-4-7-4-7-4-4-7-4-3-4-7-4-7-5-4-7-4-4-4-4-7-5-12-4-7-4-7-7-4-4-5-4-7-4-7-7-4-4-4-7-4-7-4-7-7-4-7-4-7-4-3-0-3-7-4-7-4-7-7-4-7-7-4-7-9-9-7-7-7-4-9-4-7-4-6-7-4-4-7-4-7-4-7-7-9-9-9-4-4-7
tElock 0.99	4-5-4-3-9-9-4-6-9-9-9-4-9-4-4-4-4-4-4-4-4-4-5-4-4-5-4-3-5-4-3-4-3-2-4-5-9-9-1-4-4-4-4-4-9-4-4-4-4-4-3-5
MEW 11 SE v1.1	3-3-3-5-3-7-8-5-3-3-6-4-3-4-3-1-7-7-3-3-12-12

Current experiments of the packer identification, which focuses only on the frequency of obfuscation techniques and not on their orderings, are quite preliminary; Nevertheless, BE-PUM identifies all packers that are identified by the packer signature based commercial tools, e.g., *Virus Total*, and few with the packer signature modifications are newly identified.[u1]

As additional academic activity, we organized NII Shonan meeting (a Japanese version of Dagstuhl Seminar) No.65 “Low level code analysis and applications to computer security” on March 2-5 2015 cooperated with Prof. Jean-Yves Marion (LORIA, University of Lorraine). There were about 20 participants, mostly from abroad (including 2 from Google). We are proposing the next NII Shonan meeting opportunity “Binary code analysis and applications to computer security” on March 2017.

Results and Discussion: This project developed a binary CFG generator BE-PUM, which also works as a strong disassembler of x86 binary code under presence of obfuscation techniques. We are going to open a web site (bepum.jaist.ac.jp) at JAIST to be able to test BE-PUM ability. Our ultimate goal is to classify malwares by their techniques, including the family tree of the evolutionary relationship. This project confirms that BE-PUM effectively analyzes obfuscation technique analysis, and gave us an opportunity to kick-off a broad range research on binary code analyses. There are lots of future works, and we will target on:

- Automatic generation of x86 binary emulation from natural language specification: Currently 100 x86 instructions among 1000 are emulated in BE-PUM, which are manually implemented. Compared to Windows API stub generation, x86 emulation will be more complicated, since it requires formal semantics of an x86 instruction. Based on natural language processing, ambiguity can be removed by test execution.
- Formal description of contamination techniques: 90% of malware attacks for the contamination are considered to be classified into buffer overruns. Based on disassembly results, contamination techniques will be manually observed to give their formal descriptions. In recent malware, obfuscation techniques are mostly inserted by packers. We expect that the sequences of contamination techniques in the payload of a packed code will indicate more on the evolutionary relationship.
- Model checking on disassembled code: In high-level programming languages, a backbone model for model checking is immediately given as a CFG, whereas a CFG of a binary code is not easy to generate, especially under the presence of obfuscations. BE-PUM effectively provides a pushdown model (inter-procedural CFG) of binaries. For instance, the detection of a specific Windows API call sequence (described in CTL/LTL) will indicate malicious intension.
- The family tree of the evolutionary relationship: The evolutionary relationship will be captured as a transformation relation among context free grammar representations of CFGs, which are obtained by a translation from pushdown models. This is an extremely challenging task.

List of Publications and Significant Collaborations that resulted from your AOARD supported project:

a) Peer-reviewed jounal: *None*.

b) Peer-reviewed conference:

[c1] M.H. Nguyen, T.T. Quan, M. Ogawa, Obfuscation code localization based on CFG generation of malware, 8th International Symposium on Foundation and Practice of Security (FPS 2015), Springer LNCS 9482, pp.229-247.

[c2] M.H. Nguyen, T.T. Quan, D.A. Le, Multi-threaded on-the-fly model generation of malware with hash compaction, 18th International Conference on Formal Engineering Methods (ICFEM 2016), *to appear* (November 14-18 2016, Tokyo).

c) Not peer-review journal/conferences: *None*.

d) Oral presentations at international workshops:

[o1] M.H.Nguyen, T.T. Quan, M.Ogawa, Pushdown model generation for malware deobfuscation, NII Shonan meeting No.65 “Low level code analysis and applications to computer security”, March 2-5 2015.

[o2] M.Ogawa, T.B.Ngo, Constant propagation for binary CFG rebuilt, NII Shonan meeting No.65 “Low level code analysis and applications to computer security”, March 2-5 2015.

e) Unpublished draft under submission:

[u1] M.H. Nguyen, M. Ogawa, T.T. Quan, Packer identification by semantic signature, *draft.*

f) Any interactions with industry or with Air Force Research Laboratory scientists or significant collaborations that resulted from this work.: *None.*